Teaching Foundations of Computation and Deduction through Literate Functional Programming and Type Theory Formalization

Gérard Huet

Inria Paris Laboratory

anantaṃśāstram alpaṃjīvitam
La science est sans fin, mais la vie est courte

Abstract

We describe experiments in teaching fundamental informatics notions around mathematical structures for formal concepts, and effective algorithms to manipulate them. The major themes of lambda-calculus and type theory served as guides for the effective implementation of functional programming languages and higher-order proof assistants, appropriate for reflecting the theoretical material into effective tools to represent constructively the concepts and formally certify the proofs of their properties. Progressively, a literate programming and proving style replaced informal mathematics in the presentation of the material as executable course notes. The talk will evoke the various stages of (in)completion of the corresponding set of notes along the years, and tell how their elaboration proved to be essential to the discovery of fundamental results.

1 A few general remarks about course notes

A course is a sequence of oral performances (lessons) in front of the students. The first requirement of a lesson is to arise the interest of the students in the topic under discussion, to keep their attention through appropriate histrionics, and to give them a guided tour through the course notes. This assumes that complete course notes exist, so that the students may study them and learn the material by self-study.

The easy way is to recomment a standard textbook on the subject and to follow the book, in the traditional role of a reader. This is appropriate for well-established topics at introductory or intermediate level, and does not require bleeding edge knowledge of advanced research from the part of the lecturer.

Advanced courses, at the graduate level, are another matter. Unless some researcher has recently published a comprehensive treatment of the material, the lecturer must prepare his own course notes synchronously with the teaching.

If you teach recent research of yours, the course preparation will not distract you too much from your research activity, and furthermore it opens the possibility of recruiting bright young scientists to join your team. However you will have to refrain from presenting very recent material without appropriate pedagogical treatment.

The benefit for the researcher is that this pedagogical grinding will offer opportunities for abstraction and modularity in the definitional apparatus, while well-chosen examples will illustrate the material in its possible use in other contexts.

2 A short history of my own course notes

2.1 Logic

I started to teach logic for my graduate course on foundations of informatics, back in the 70's. There were many textbooks on mathematical logic, all with the same structure. First classical propositional logic. Then first-order predicate calculus. Together with Gödel's completeness theorem, looking like stating the obvious. Then Gödel's incompleteness theorem, just as if it was a deliberate attempt to utterly confuse students with ambiguous terminology. Then a bit of arithmetic, possibly extended with recursive function theory. It would usually end up with rudiments of axiomatic set theory. That is, ending by pretending to define what was assumed known earlier in Tarskian semantics. The whole thing was just a complicated vicious circle claiming to be Logic.

Equality was slipped under the carpet, usually presented as just one specific binary predicate verifying a bunch of axioms. Intuitionistic logic was alluded to in a footnote, as some weird variation, as well as modal logic. No status was given to definitions in the object language. Often natural deduction was omitted in favour of Hilbert's formulation, hiding the fact that the deduction (meta)theorem was the algorithm of lambda abstraction defined recursively over combinator trees. It was left as an exercice to prove that $A \Rightarrow A$, as if complex reasoning was needed to prove evidence, making a joke of the total endeavor.

The only decent introduction was "Notes on Logic", a small monograph[24] written as logic course notes in 1964 by Roger Lyndon, an algebraist. He presented under a succinct account a good selection of the main topics in a lucid fashion, with consistent terminology. For instance, completeness of the proof system was renamed as adequacy, keeping the name "completeness" for a property of logical theories. I felt a lot of admiration for a logic outsider, using his algebraic training to clean up the material in order to present a clear presentation for his students. Insiders often present unpalatable presentations of their specialty, use routine notation without questioning the standard terminology, and tend to indulge in ruminated broccoli.[1]

Lyndon's illuminating booklet was the main inspiration for my first set of course notes, "Initiation à la Logique Mathématique"[4], which I taught at Université d'Orsay (also known as Paris XI, Paris-Sud, Paris Saclay, etc.) in 1976-77. My main innovation was to introduce "camembers" as graphical renditions of first-order theories.

In order to approach the subject of classical logic from a computer scientist point of view, I wrote a supplement to these course notes in 1978, entitled "Démonstration automatique en logique de premier ordre et programmation en clauses de Horn"[5], developing term structures, unification, resolution theory, then Prolog. These notes came to be well-known for their final section, showing how to use Prolog to parse the sentence "La stupidité naturelle encourage l'intelligence artificielle". This was my first foray into non-politically correct pedagogical stratagems: Mettre les rieurs de son côté.

2.2 Rewriting

In 1976 I started to get interested in rewriting, and set to write course notes entitled "Règles de simplification". In order to present the Knuth-Bendix method[23] in a general way,

[1] Gilles Kahn called ruminants the authors of minor improvements or variations of known material; Jean-Yves Girard calls broccoli the un-necessarily bloated presentation of ad-hoc axiomatisations, with the usual un-surprising adaptation of the adequation theorem in all its tedious but routine details.

abstracting from the first-order term structure, I had divided the course into 2 parts; the first one was just some relational algebra. I coined the term “confluence” for what was known in various guises as the “diamond property”, the “Church-Rosser property”, the “parallel moves lemma”, etc. For termination, I coined the term “nœtherian”, and gave an inductive characterizaton through reachability, coined “Nœtherian induction”. This allowed me to prove Newman’s lemma in two easy steps, to be compared to heavy topological considerations in the original version.

For term structure, I showed how to prove confluence from the confluence of critical pairs, avoiding any termination condition, and thus generalizing from Knuth and Bendix who used the existence of normal forms, an irrelevant consideration. All these improvements are directly issued from pedagogical considerations of modularity and good choice of primitive notions with careful terminology. The resulting course notes improved the state of the art, to the point of providing with almost no change a research communication to JACM, my “Confluent Reductions”[6] paper.

This graduate course at Orsay University was concrete enough to attract the attention of the best student in the class and programming wizard, Jean-Marie Hullot, who set to put the constructive bits into actual LISP software, the KB completion platform. Now superposition, unification, narrowing, rewriting were made alive as real programs, providing a toolkit for further research, as well as a concrete platform for teaching the material. Thus the extension to inductive theories was suggested by experimentations, giving a basis to “inductionless induction”[21], and further analysis on the KB completion result gave rise to the “unfailing completion” method[19]. All this fine-tuning could now be presented as a textbook survey of the state of the art “Equations and Rewrite Rules”[22].

2.3 From λ-calculus to category theory

Lambda-calculus, like unification, is one of these recurring themes that I met during my career again and again at unsuspecting turns. Many times in my scientific careerI got this oceanic feeling of eternal return. First I had the luck to discover early the work of Jim Guard and his mathematician colleagues at the Applied Logic Corporation in Princeton[2], and thus to investigate unification in simply typed lambda-calculus. Which led to have a glimpse at non-trivial formal mathematics with Church’s simple theory of types. Then I was fortunate to learn the syntactic theory of pure λ-calculus from my Inria colleague and friend Jean-Jacques Lévy. This connected with Landin’s Iswim, Milner’s ML and functional programming.

Then I encountered Girard’s systems F and Fω, and this connected to Reynold’s polymorphic λ-calculus. Furthermore, through the Curry-Howard-de Bruijn’s isomorphism, λ-calculus appeared as the ubiquitous algebra underlying Gentzen’s natural deduction. Now I knew that the Graal of our quest was λ-calculus. Pierre-Louis Curien pointed out to me Lambeck’s axiomatisation of CCCs, which gave a categorical outlook to λ-calculus in an equational setting, connecting with my rewriting interests. Much later I understood from the work of Montague that λ-calculus was also the skeleton of linguistic productions.

I had resisted till then to learn category theory, despite the hints of Gordon Plotkin and Gérard Berry. I was dissatisfied with the current textbooks on category theory, and decided to teach it in my own way. This lead to the booklet “Initiation à la Théorie des Catégories”[18], used in my course at Université Paris 7 within the graduate curriculum “Fonctionalité, Structures de Calcul et Programmation” from 1983 to 1985.

This booklet was written in MacWrite, the word processor of the original Macintosh. This software, coupled with MacDraw, was remarquably suited to produce quality documents including graphics. The various typographical facilities were sufficient to express mathematical notation in a satisfactory manner. I paid utmost attention to using fonts in a systematic manner for uniform notation. Boldface was used for categories and functors.

I revealed a few secrets of category theory reasoning, where commuting diagrams are actually sorted equations between arrows, represented on a shared dag datastructure: *Le raisonnement diagrammatique permet de faire simplement des preuves équationnelles compliquées, en utilisant une structure de graphes partagés pour représenter les termes et les preuves.* This intuition arose from my acquaintance with equational logic and rewriting on one hand, and the Curry-Howard isomorphism on the other. My acquaintance with the relationship between exponentiation and lambda abstraction, from the works of Lambek and Pierre-Louis Curien, motivated me to use the terminology “catégorie calculatoire” as a computer-science palatable alternative to the cryptic official name Cartesian Closed Category. I showed the connection to logic by interpreting exponentiation as intuitionistic implication, a relief after those years of teaching classical logic to computer-science students with no better motivation that “this is the way logicians explain logic”. The terminology “Computable Category” was then justified by exhibiting λ-calculus as their internal language. With the added satisfaction to see de Bruijn’s indices[1] percolating naturally from the mathematics.

I got carried away, and included a section “Compléments” with more advanced topics, such as adjunction, in order to exhibit the 2 magic steps that lead from Diagonal to Product to Exponentiation, and how one can create both Deduction and Computation structures from just duplication, using twice the Galois connection magic wand. Finally, I used rewriting theory at the level of natural isomorphisms, and showed how the Mac Lane and Kelly coherence conditions could be mechanically computed by the Knuth-Bendix completion procedure.

This is a perfect exemple of how research (in term rewriting) contributed to teaching (category theory), leading to new insights and results in the topic taught. I even presented this coherence conditions compilation to an audience of category theorists in Denver[7]. This provoked such wrath in William Lawvere that I decided to avoid such insane gatherings in the future, and never published the result!

This leads me to a word of caution concerning lecture notes. When they include original unpublished material, they are easy prey for unscrupulous colleagues, who will shamelessly pillage your definitions, proofs and exemples without proper quotation, on the rationale that since this is course notes material, it must be well known...

2.4 From category theory to λ-calculus

The last booklet in the series of MacWrite French course notes was started in 1985 as “Initiation au λ-calcul”[9]. I taught this course material over many years at Université Paris 7. It profits from the various aspects of the fundamental functional algebra, from functional programming to proof theory to category theory. It is an uncompromising presentation, where all definitions are presented formally as recursions over terms in contexts, with variables expressed through de Bruijn’s indices. This allows to get rid once and for all of strings used as names, with α-conversion some kind of magic incantation invocated to avoid capture of free variables by local binders during substitution. α-conversion was not only un-necessary and imprecise, but it perverted the whole spirit of exact formalisation of this complex material with a veil of vagueness that transformed any demonstration into a hand-waving exercice impossible to validate formally.

Admittedly, λ-terms with de Bruijn indices is not for human to contemplate, being the analogue of machine language with respect to high-level programming languages. Thus, there was a necessity to distinguish between abstract syntax using indices for the theory and concrete syntax using names to be able for a human to run exemples and retrieve results. Thus α-conversion was placed at its proper place of man-machine interface, without theory corruption. I thus needed to give to my students all the tools to go back and forth between the two aspects, and this lead to a fundamental ontological requirement: having executable course notes. Thus every mathematical definition was turned into a recursive definition in a functional programming language, and it was only fitting for me to use Caml, the "categorical abstract machine language", our ML implementation, to serve as meta-language of the course notes. The ML programs were available to the students, and they could thus check all details of long β-reduction computations, with no risk of tripping over the names proliferations and the jumbling of parentheses. This also lead to spectacular improvements to such notions as residuals, who were just computed out as extra decorations traced along a derivation.

When taken seriously, the idea of executable course notes merges with the important concept of literate programming. Since the most important source of the course notes is a library of algorithms, we may now view the lecture notes as proper presentation of the library. This is the concept of *literate programming* as proposed by Don Knuth: the lecturer reads good literature, that explains the concepts underlying the programs. The notes give the meta-theory as mathematical explanation of the programs properties (such as termination), justifying in turn their use as mathematical definitions within a corpus of constructive mathematics. This is the way Informatics goes hand in hand with Mathematics. We package the program sprinkled with its verification conditions into some kind of pudding, and its proof of correctness is in the pudding if the verification software type-checks it. Actually preparing the type theory pudding yields the executable program as a side-effect. We are after *programs correct by construction.*

3 Functional programming as a mathematical notation

3.1 The CMU notes

In 1985, I got an invited professorship at Carnegie-Mellon University and went to work in the Theory Group headed by Dana Scott. I taught one graduate course, entitled "Formal Structures for Computation and Deduction"[20]. I took this opportunity as a chance to test my executable course notes idea on a unified view of formal structures for logic and computing defined in ML. This encompassed rewriting theory, including recent results on sequentiality obtained with Jean-Jacques Lévy, λ-calculus, of course, natural deduction, type theory, and even a presentation of polymorphic and dependent type theory, through the Calculus of Constructions that was just issued from Thierry Coquand's PhD thesis. The last chapter on induction, which was in itself a concentrated survey on various facets of this important topic, ended up to be published in [11].

The transition from course notes to literate programming documentation is specially striking concerning type theory. The basic axiomatization of the Calculus of Constructions was the basis for the first implementation of the Coq proof assistant in Caml, which lead to the publication of the Constructive Engine research paper [13], where a (slightly simplified) Caml library issued from the actual implementation of Coq was used as mathematical presentation of its proof checker.

Writing the course notes ahead of the lectures proved to be a taxing endeavor. It involved quite a bit of hacking, since I had to build the ML libraries ahead of the physical class. Making the course notes executable in a context of programs under development and revisions involved some hacks to link the Caml algorithms to my LaTex sources, then invoking the code from Emacs to textually substitute the resulting outputs to my TeX source in verbatim mode. This hack actually was made the basis to a requirement of the literate programming style. In order to present transparently the material, the course had to develop all the notions in an order consistent with the sequential execution of the libraries. Thus even basic algorithms had to be presented at the proper moment. This way, the user (the student) could start with no other prior knowledge than the semantics of ML.

ML was quickly presented as the bleeding edge of recursive definitions, and the fiction that its evaluation was just the usual bottom-up computation of arithmetic expressions could be maintained until the λ-calculus chapter where its proper definition could be finally exhibited as a precise β-reduction tactic. Many programmers are surprised at the standard ML style of primitive recursion over inductively defined data structures. At least, at the time. Specially diehard C fanatics, who can't fathom using data patterns as left values, not to speak of closures encapsulation of code in data. Logicians in the class were not surprised, knowing Kleene's presentation of recursive functions.

I would like to point out my use of `where` clauses, in complement of the `let`. `where` clauses are very important in view of relaxing the sequentality condition, at least locally. I would typically write:

```
iter process (collect_all_candidates input)
  where process candidate = interesting_algorithm
  and collect_all_candidates x = ugly_complex_computation
```

which allows to focus on `interesting_algorithm` and reject to later consideration the complicated part, even though in the flow of control it is executed first. This made the oral teaching much smoother. My obstinate use of `where` clauses, through use of the Camlp4 preprocessor acting as ML parser for a well-structured uniform syntax, is a subject of derision by colleagues and students from the Gallium team. Thay had long denegated its usefulness, deprecated its usage, and finally excised its syntax from Ocaml as superflous, being redundant with `let`. So let me take this occasion to reiterate: "If we believe that ML should be the publication language of algorithms presented in literate programming style as executable course notes for oral lectures, the where clauses ought to be preserved for pedagogical purposes".

Actually, I could not quite sustain the hacking rythm during the semester-long course. My last program in the notes is `kb_completion`, a full fledged Knuth-Bendix completion procedure with delayed failures, and its application to the synthesis of the canonical group presentation. Xavier Leroy told me that this program is still running daily in the Ocaml test bench.

Thus ended this attempt at writing executable course notes for computation and deduction. A copy was edited in May 86 as 1st Edition of something that never got published, except as samizdat on the Web. It got reprinted as teaching material for the Marktoberdorf Summer School on Logic of Programming and Calculi of Discrete Design in August 1986[8]. I also provided shorter course notes in a more conventional style called "Deduction and Computation". This last document evolved into a more substantial document "A Uniform Approach to Type Theory", which I used as course material in 1988 for a workshop on Logical Foundations of Functional Programming organized as UT Austin.[14]

3.2 λ-calculus theory in functional programming

Meanwhile, my French course notes “Initiation au λ-calcul” had been extended to cover typed λ-calculus, starting with conjunctive types along Krivine’s presentation. It went into polymorphism, dependent types, and ended with the Calculus of Inductive Types, giving a complete introduction to Coq. The last MacWrite document was completed in 1991, ending my French trilogy on foundations of computation. An editor expressed interest in publishing the series, but wanted a 4th volume, which never was written, and thus most of this material slipped into oblivion. The pure λ-calculus first part of “Initiation au λ-calcul” was published in a Marktoberdorf volume[10] in 1991.

With the model of executable course notes I had started at CMU, I set to restart the effort systematically, at least for pure λ-calculus, in English, and with full ML axiomatization. At the end of 1986 I took a mini-sabbatical as Invited Professor to the Asian Institute of Technology in Bangkok for 2 months. In this paradisiac environment, I managed to develop completely the theory of Böhm trees necessary to prove Böhm’s separation theorem.

This was not trivial, because these are potentially infinite structures which one would think must be built by co-induction. Then the encoding of variables had to be invariant by η-expansion, something that is not true of the de Bruijn treatment, but actually works in the inverse notation, left to right in the prefix of a head normal form. But I wanted to keep de Bruijn indices to refer to the levels of hnf approximants of the Böhm tree. Thus I came up to the notion of a variable occurrence encoded as a pair of indices, the first one backward indexing the levels à la de Bruijn, the second one numbering bound variables left to right, in the forward direction this time. In other words, the display mechanism of Algol 60 emerges from the theory. Then I got the idea of unfolding a lambda term progressively as successive Böhm tree approximants keeping frozen a non-evaluated term as a future.

```
type var = Index of (int * int)
and bohm = [ Hnf of (int * var * bohm list)
           | Future of term
           ];
```

With this definition, everything fell in place. I could program the Böhm out technique, and prove a general separation theorem in its full generality, Böhm’s theorem being the special case for normalisable terms. The Böhm out technique leads to the natural definition of what I coined *schizophrenic* head variables. Schizophrenic variables have occurrences on the separating path that differ in the direction of their argument. Thus you cannot just substitute for them a uniform projection combinator, you have to linearize first by substituting for them the pairing combinator, that introduces a fresh variable at every occurrence. Thus by application of the successive terms of the discriminating context either one level collapses on the path by projection, or linearisation decreases the schizophreny of the situation while keeping the depth constant. This is the key to define the lexicographic ordering that proves termination.

Teaching separability then becomes easy. You point out the difficulty by using striking terminology from mental disorders which turns out to be self-explanatory. The schizophreny reflects the perplexity of the student trying to directly project along the separating path. This perplexity is transfered to the relevant head variable personified as a mentally disturbed individual, giving a tragedi-comical turn to the technical problem that leaves an indelible imprint in the mind of the student.

This kind of gimmick, using creative terminology and notation, is actually a very effective pedagogical device. I also developed over the years specific histrionics to give dynamics to definitions. I came to teach confluence with a simple gymnastics exercise that makes explicit in a dramatic way the quantifier alternation which is essential to the proper use of the concept. I even went to explaining λ-calculus as some kind of social description of an out-of-space population. Males were applications, females were abstractions. There was a third kind of unsexed beings called nymphs, standing for variables. They wore a little hat with a counter, de Bruijn indexing their mother. β-conversion was a gory reproduction affair. I would probably had been radiated from the University and perhaps even arrested if I had taught this material in the USA. French students loved it, irrespectively of their gender.

The formal development of separability was finally published as a research paper in Theoretical Computer Science[16] in 1993, with verbatim ML formalisation. This style became my standard way of writing constructive mathematics, although it is not always easily supported by scientific journals editors, who often prefer the usual hand-waving without-loss-of-generality approximation socially accepted as rigorous mathematical writing.

The separability theorem is the crucial key to understand the (meagre) degrees of freedom in constructing models of pure λ-calculus, since quotienting Böhm trees is limited to η, limiting effectively possible choices to two interesting models.

Now I had all the ingredients to write executable course notes on pure λ-calculus, and to develop with it recursion theory in an elegant fashion. This lead to the *Constructive Computation Theory* notes, which I used as teaching material in many courses, lastly at Master Parisien de Recherche en Informatique in Paris in 2004. It is still available as a samizdat on the Web[15].

4 Type theory as a specification language

Then came the second epistemological U-turn. I was teaching that proofs were actually annotations of functional programs "correct by construction". But my proofs of the meta-theory were done in the good old hand-waving way. I had to taste my own medicine, so to speak, and redo the whole effort in type theory, with formal proofs verified by Coq. Logically, the functional programs would be extracted as the constructive contents of those proofs.

I set to the task of reworking the syntactic theory of β-reduction as a Coq development. The uncompromising commitment to de Bruijn indices in the terms representation lead to a painful development of the arithmetic of relocation during substitution. Once this hurdle was overcome, which I claim is still both simpler and more efficient than having to manage whatever naming mechanism, a spectacular simplification was obtained with the Substitution Lemma. This fundamental distribution property of substitution was now stated rigourously as a mathematical identity for the first time, after all these years of α-conversion hand-waving.

Similarly, using annotations to keep track of residuals in an enriched structure, it appeared that the commutation of residuals was just confluence of the marked terms. This fundamental property was expressed as the Commutation Theorem:

Commutation Theorem. If U_1 and V_1 (resp. U_2 and V_2) are compatible sets of redexes:

$$(V_1/U_1)\backslash(V_2/U_2) = (V_1\backslash V_2)/(U_1\backslash U_2)$$

writing V/U for the substitution of V in U and $U\backslash V$ for the residuals of U by V.

Look, Ma, no de Bruijn index, they got eliminated in the low-level lemma shuffling!

From this fundamental theorem resulted Lévy's cube lemma, which shows that confluence of λ-calculus computations is not just a syntactic coincidence as in the general term-rewriting situation. It is a deep algebraic commutation property of derivations quotiented by the natural equivalence induced from residual preservation. In categorical terms, the derivations category admits push-outs.

This research led to the publication of “Residual theory in λ-calculus”[17] in 1994. This marked the path to follow: literate proving. And this showed that the admittedly enormous effort put in steering a proof assistant to formally establish the properties of our formalisms has a fantastic reward: the right definitions just jump out of the development as necessary abstractions, presenting the material in novel and elegant setting.

Thus my course notes have been receding in an infinite regress procrastination in an elusive search for the perfect formulation. They will never be completed, but younger scientists are taking the lead to continue the task. Nowadays it is considered standard to present logic and programming notions with actual open-source programs and formal proof developments. The material of my course notes is now available in better presentations than my primitive literate programming style, witness e.g. [25, 3].

5 Conclusion

I have presented successive attempts at writing rigorous presentations of the fundamental structures of computation and deduction, using firstly functional programming, and then type theory. This sort of presentation has an pleasingly esthetic reflexive character: we eat of our own pudding, and build the foundations of our own mathematical notation.

The history of my course notes and the history of my research publications are co-extensive. In a sense, all my research was pulled by my teaching imperatives. Also, the search for perfection in the theoretical material turned into some quest of the ultimate beauty of technical concepts. Amazingly enough, this esthetics imperative does not come to the detriment of the efficiency of the computational frameworks we are designing, quite to the contrary. This motivated me into daring expressing the idea as a provocative manifesto in my CMU notes: *It is our thesis that formal elegance is a prerequisite to efficient implementation.*

Acknowledgment

This paper was prepared as material for an invited talk at the 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016) in Porto, Portugal, in June 2016. It did not make it to the published volume, though.

References

1 N. G. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34:381–392, 1972.

2 Jim Guard. Automated logic for semi-automated mathematics. Technical report, Applied Logic Corporation, 1964. Scientific Report, AFCRL contract.

3 John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

4 Gérard Huet. Initiation à la logique mathématique. Notes de cours du DEA d’Informatique, Université Paris-Sud, 1977.

5 Gérard Huet. Démonstration automatique en logique de premier ordre et programmation en clauses de Horn. Notes de cours du DEA d’Informatique, Université Paris-Sud, 1978.

6 Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27,4:797–821, 1980.

7 Gérard Huet. Canonical basis of commuting diagrams. Application to the mechanical synthesis of coherence conditions. Slides of a presentation at a category theory meeting at the University of Colorado in Boulder. Available at URL `http://yquem.inria.fr/~huet/PUBLIC/Boulder_categories_slides.pdf`, 1987.

8 Gérard Huet. Deduction and computation. In Manfred Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 305–342. Springer Verlag, 1987.

9 Gérard Huet. Initiation au λ-calcul. Notes de cours du DEA "Fonctionalité, Structures de Calcul et Programmation, Université Paris VII, 1987.

10 Gérard Huet. Introduction au λ-calcul pur. In Friedrich Bauer, editor, *Logic, Algebra and Computation*, pages 153–200. Springer Verlag, 1987.

11 Gérard Huet. Induction principles formalized in the calculus of constructions. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 205–216. North Holland, 1988.

12 Gérard Huet. Cartesian closed categories and lambda-calculus. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 7–23. Addison-Wesley, 1989.

13 Gérard Huet. The Constructive Engine. In R. Narasimhan, editor, *A perspective in Theoretical Computer Science. Commemorative Volume for Gift Siromoney.* World Scientific Publishing, 1989.

14 Gérard Huet. A uniform approach to type theory. In Gérard Huet, editor, *Logical Foundations of Functional Programming*, pages 337–397. Addison-Wesley, 1989.

15 Gérard Huet. Constructive computation theory. Course Notes, DEA Informatique, Mathématiques et Applications, Paris VII. Available at URL `http://yquem.inria.fr/~huet/PUBLIC/CCT.pdf`, 1992.

16 Gérard Huet. An analysis of Böhm's theorem. *Theoretical Computer Science*, 121:145–167, 1993.

17 Gérard Huet. Residual theory in λ-calculus: a formal development. *J. Functional Programming*, 4,3:371–394, 1994.

18 Gérard Huet. Initiation à la théorie des catégories. Notes de cours du DEA "Fonctionalité, Structures de Calcul et Programmation, Université Paris VII, 2nd edition, 1987.

19 Gérard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *J. of Computer and System Sciences*, 23,1:11–21, I981.

20 Gérard Huet. Formal structures for computation and deduction. Course Notes, Carnegie-Mellon University. Available at URL `http://yquem.inria.fr/~huet/PUBLIC/Formal_Structures.pdf`, May 1986.

21 Gérard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. *J. of Computer and System Sciences*, 25,2:239–266, I982.

22 Gérard Huet and Derek Oppen. Equations and rewrite rules: a survey. In Ronald Book, editor, *Formal Languages: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

23 Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.

24 Roger Lyndon. *Notes on logic.* Van Nostrand, 1966.

25 Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.